h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

– Fachbereich Informatik –

Transactionmanagement for distributed routers in an IP-backbone

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von

Dominik Mandok

Matrikelnummer: 769382

Referent	:	Prof. Dr. Andreas Heinemann
Korreferent	:	Kai Naschinski

DECLARATION

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 07. März 2024

Dominik Mandok

ABSTRACT

This thesis addresses the significant challenge of ensuring consistent configuration updates within IP backbone networks, specifically focusing on the routers used by *Deutsche Telekom*. It proposes a novel approach by applying transaction management principles, typically seen in database management, to network configurations. This involves creating an autonomous system that ensures either the complete success or rollback of updates, thus maintaining network stability and reliability.

The thesis explores the concept of Backbone as Code for automating router configuration updates and critically analyzes current methods and their inherent limitations, such as manual configurations. It also introduces the application of Software Defined Networking to simplify and automate network management. The main focus is on adapting transaction management to network settings, emphasizing the necessity for atomicity, consistency, isolation, and durability in network updates.

A practical aspect of the thesis includes developing a virtual router lab and a script to manage configuration changes, demonstrating the application of the proposed transaction management system. The implementation showcases the potential of this approach in enhancing network management efficiency, reducing human error, and ensuring consistency in large-scale IP Backbone environments. The thesis addresses a current gap in network management and lays the groundwork for more robust and reliable network systems.

ZUSAMMENFASSUNG

Diese Arbeit geht der Herausforderung nach, konsistente Konfigurationsaktualisierungen in IP-Backbone-Netzwerken sicherzustellen, wobei er sich speziell auf die von der *Deutschen Telekom* verwendeten Router konzentriert. Die Arbeit schlägt einen neuartigen Ansatz vor, indem sie Prinzipien des Transaktionsmanagements, die üblicherweise im Datenbankmanagement zu finden sind, auf Netzwerkkonfigurationen anwendet. Dies beinhaltet die Schaffung eines autonomen Systems, das entweder den vollständigen Erfolg oder das Zurücksetzen von Updates gewährleistet und so die Stabilität und Zuverlässigkeit des Netzwerks aufrechterhält.

Die Arbeit erforscht das Konzept von Backbone as Code zur Automatisierung von Router-Konfigurationsaktualisierungen und analysiert kritisch die aktuellen Methoden und ihre inhärenten Einschränkungen, wie manuelle Konfigurationen. Zudem wird die Anwendung von Software Defined Networking eingeführt, um das Netzwerkmanagement zu vereinfachen und zu automatisieren. Der Hauptfokus liegt auf der Anpassung des Transaktionsmanagements an Netzwerkeinstellungen und betont die Notwendigkeit von *Atomicity*, *Consistency*, *Isolation* und *Durability* bei Netzwerkaktualisierungen.

Ein praktischer Aspekt der Arbeit beinhaltet die Entwicklung eines virtuellen Router-Labors und eines Skripts zur Verwaltung von Konfigurationsänderungen, was die Anwendung des vorgeschlagenen Transaktionsmanagementsystems demonstriert. Die Implementierung zeigt das Potenzial dieses Ansatzes, die Effizienz des Netzwerkmanagements zu verbessern, menschliche Fehler zu reduzieren und Konsistenz in groß angelegten IP-Backbone-Umgebungen sicherzustellen. Die Arbeit adressiert eine aktuelle Lücke im Netzwerkmanagement und legt den Grundstein für robustere und zuverlässigere Netzwerksysteme.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABKÜRZUNGSVERZEICHNIS

2PC	Two-Phase Commit
AR	Access Router
AS	Autonomous System
ASN	Autonomous System Number
BBaC	Backbone as Code
BGP	Border Gateway Protocol
BR	Backbone Router
CLI	Command Line Interface
DHCP	Dynamic Host Configuration Protocol
EGP	Exterior Gateway Protocol
EIGRP	Enhanced Interior Gateway Routing Protocol
IaC	Infrastructure as Code
IGP	Interior Gateway Protocol
IS-IS	Intermediate-System-to-Intermediate-System
ISP	Internet Service Provider
KPI	Key Performance Indicator
LLT	Long Lived Transaction
LSP	Label Switched Path
MPLS	Multiprotocol Label Switching
NDO	Network Domain Orchestrator
OS	Operating System
OSPF	Open Shortest Path First
RAR	Remote Access Router
RIP	Routing Information Protocol
SDN	Software Defined Networking
SPF	Shortest-Path-First
SQL	Structured Query Language
SSoT	Single Source of Truth

Part I

THESIS

1 INTRODUCTION

In the rapidly evolving field of information technology, emphasizing efficiency and reliability is critical to innovation. Businesses and service providers are increasingly challenged by growing data volumes and complex network demands [9, pp. 5–6]. In this context, the importance of automation in addressing these challenges is clear. Automation significantly enhances productivity by reducing the need for employees to perform repetitive tasks, and it ensures high precision and reliability in operations. This is particularly evident in IT infrastructure management, where the concept of Infrastructure as Code (IaC) is transforming the approaches used for configuring and maintaining virtual machines, servers, and networks.

At *Deutsche Telekom*, a German Internet Service Provider (ISP), automation is a daily topic. This thesis is also part of a currently running project called Backbone as Code (BBaC) at *Deutsche Telekom*. The project's name was selected because of the similarities to IaC. The goal is to - as far as possible - develop an autonomous system that handles configuration changes of the routers inside the IP backbone, which is the core network of the ISP handling all the traffic of millions of customers in Germany (more in section 3.2). The system has to perform updates on multiple operating systems by different vendors. A very important subtask in updating the routers that power the IP backbone of the *Deutsche Telekom* is to ensure that updates are either fully finished and running or, in case of a failure, rolling those already made changes back and resetting the network to its previous running state. The needed network update consistency has to be provided by the developed system, and as a part of this thesis, a prototype of such a system will be described and built (chapter 5).

One possible solution for this requirement could be transaction management, as it's a well-known topic in the field of (distributed) database management. It coordinates multiple systems to reach the next consistent state [13]. A database transaction begins when a program issues a command, either explicitly or implicitly, through data modification. The program then performs various operations such as reading, inserting, updating, or deleting data. Throughout this, the database maintains integrity by ensuring all functions within the transaction are atomic - they either all succeed or fail together. The database checks for conflicts with other transactions during the validation phase, ensuring consistency and adherence to ACID[1] properties [13, p. 290]. If a conflict arises, the transaction may be rolled back or delayed. Once validated, the transaction reaches the commit stage, where the program issues a *Commit* command to make changes permanent. The database writes these

1 Atomicity, Consistency, Isolation, Durability

changes to disk, marking the transaction's completion and ensuring durability. If the transaction is successful, the database reaches a consistent state with the new data accessible for other transactions. If any operation fails or a *Rollback* command is issued, the database undoes all operations, reverting to its state before the transaction to maintain consistency.
Especially distributed transactions using the Two-Phase-Commit protocol (section 4.2.1) or sagas (section 4.2.2) will be examined in a designated chapter later on.

This principle also applies to configuration management within an IP backbone, which can help ensure consistency and reliability. Changing multiple router configurations simultaneously is prone to errors that must be handled since one misconfiguration may lead to connectivity issues spanning an extensive network. [9, p. 6]

1.1 BACKBONE AS CODE

The idea behind the project BBaC is to introduce a new way of updating router configurations inside the IP backbone of the *Deutsche Telekom*. While introducing this new configuration method, currently used methods like manual configuration over the Command Line Interface (CLI) or tools like the Network Domain Orchestrator (NDO) (section 3.3) should get replaced since one goal is to create a Single Source of Truth (SSoT) for the routers' configurations. The following image shows a possible architecture:

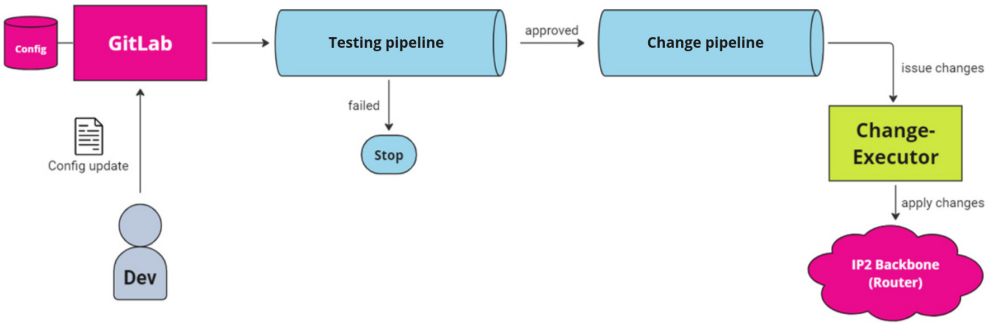

Figure 1.1: BBaC architecture

In this architecture, the SSoT is the GitLab instance of the *Deutsche Telekom*. It holds the current configurations as well as historical versions. When changes are uploaded, they get tested and applied as long as the tests don't fail. If the tests do not fail, but the change fails in the IP backbone, the change executor has to roll any written changes back to the previous consistent state so that the least possible problems occur in the live network. This architecture provides a centralized place to look up running configurations without having to access multiple different systems or pull the configuration from the actual network devices. The change process is also automated and centralized so that any human error is minimized or at least compensated for by automated testing and control mechanisms.

This thesis looks into the testing and changing processes of such a system. As the main focus lies on the "Testing pipeline" step of the architecture displayed in fig. 1.1, a prototype of a virtual router lab will be built (section 5.1) as well as a script to execute and rollback changes (section 5.2).

2

RELATED WORK

"Principles of transaction-oriented database recovery" [13] is a paper "describing different transaction-oriented recovery schemes for database systems in a conceptual [...] way". It includes a description of the four ACID properties of a database: Atomicity, Consistency, Isolation, and Durability. They are the critical indicators for the database system's quality [13, p. 290]. In combination with transactions, these properties might be a way to introduce consistent updates in an IP backbone context (chapter 4).

Harrabi et al. [14] discuss integrating NFV and SDN in MPLS networks, highlighting flexibility and efficiency. Enck et al. [10] present the PRESTO system for large-scale network configuration, emphasizing modular and automated management. Cal et al. [4] introduce GSAT for auditing router configurations, focusing on structural analysis for scalability. Birk et al. [2] explore SDN implementation in ISP backbones, advocating for an evolutionary approach to integrate SDN with traditional technologies for optimized traffic management and service provisioning. These four papers helped gather insights into network efficiency, configuration management, and integrating new technologies into network infrastructures like large ISP backbones, collectively exploring network management and optimization advancements.

"Software-Defined Networks: A Systems Approach" [17] by Peterson et al. delves into Software-Defined Networking (SDN), presenting it as a critical transformation in internet technology where control shifts from traditional hardware to cloud-based software. It clarifies SDN beyond industry hype, describing it as a scalable system on commodity hardware. The book challenges the conventional protocol stack view, introducing a new software perspective supported by hands-on exercises. It emphasizes open source, collaborating with organizations like the Open Compute Project and Open Networking Foundation, and covers various industry interpretations of SDN. "An Introduction to Computer Networks" [8] by Doral has a chapter on SDN, emphasizing its transformative role in networking. It highlights the shift from traditional hardware-centric networks to flexible, software-controlled systems using commodity hardware. Central themes include the importance of SDN controllers and OpenFlow switches in managing network traffic and enabling enhanced control. Both texts underscore the adaptability of SDN in various applications, from server load balancing to managing complex network topologies.

"Human error" [20] by Reason discusses strategies for managing errors in high-reliability systems. It explores different perspectives on error, from cognitive theory to practical application, underlining the need for error-tolerant

designs. The book aims to provide insights for a diverse readership while addressing error categorization, mechanisms, and consequences, focusing on understanding and mitigating errors in complex, high-risk technologies.

"Sagas" [11] by Garcia-Molina and Salem introduces Sagas as a method for handling long-running transactions through a series of interdependent yet independently commitable transactions, using compensations for error handling. "Theoretical Foundations for Compensations in Flow Composition Languages" [3] by Bruni et al. extends this concept, providing a theoretical framework for compensations in process composition languages, focusing on maintaining system consistency in complex transaction scenarios.

3

BACKGROUND

3.1 NETWORKING BASICS

A fundamental part of networking is an operation called "routing," which is performed by routers. Routing means deciding where to send an incoming packet not addressed to the router. These decisions are made manually by statically typing in the forwarding rules for the router or automatically by setting up routing protocols, which determine the best paths through a network. Routing protocols use different algorithms for finding the best path through a network. The main difference is about which network nodes the algorithm considers: There are Link-State routing protocols like Open Shortest Path First (OSPF) or Intermediate-System-to-Intermediate-System (IS-IS), which collect information about the whole network and use Shortest-Path-First (SPF) algorithms like Dijkstra to find the shortest path, and there are Distance-Vector routing protocols like Routing Information Protocol (RIP) or Enhanced Interior Gateway Routing Protocol (EIGRP), which know their neighbors and their neighbors' shortest route to a destination to calculate the overall shortest path [8, p. 291][9, pp. 43–45]. After calculating and storing the fastest paths to the reachable networks, the routers can forward packets to the wanted destination.
This kind of routing is usually done inside extensive networks like Carrier backbones 3.2, Peering-Points, or bigger company networks, whereas customer routers at home often have their home network and a default route configured that points to the next router in the access network of their ISP.

The following section focuses on the circumstances inside the IP backbone of an ISP.

3.2 IP BACKBONE

The IP backbone, a critical component of large-scale ISPs networks, is a complex and multi-layered structure that enables global Internet connectivity. It represents the core network segment, distinguished from the metro and access network segments, playing a central role in data movement across the Internet. [9, p. 19]

A key element of the IP backbone's structure is the distinction between Access Routers (ARs) and Backbone Routers (BRs). ARs are the initial point of contact for customer equipment, effectively serving as gateways to the broader Internet. These routers are either colocated with backbone routers or exist remotely as Remote Access Routers (RARs). Access routers connect to BRs, the main routers within the core IP layer. Usually, the ARs are *dual-homed*

to two BRs to ensure high service availability through redundancy. This hierarchical structure ensures efficient routing and management of data traffic across the network. The Internet Engineering Task Force uses similar terminology, referring to these as Customer-Edge equipment, Provider-Edge routers, and Provider routers. [9, p. 34]

Multiprotocol Label Switching (MPLS) plays a transformative role within this IP backbone architecture. MPLS, introduced in the late 1990s, allows for more efficient and flexible routing within these backbones. By creating Label Switched Paths (LSPs), MPLS enables data packets to be routed through paths not strictly defined by the shortest route, as typical in conventional IP routing. This approach allows for dynamic traffic engineering, which allows rerouting traffic away from congested links to less busy ones, thereby optimizing network performance and reliability. [9, pp. 49–52]

In the IP backbone, the relationships between routers are defined by adjacencies, which are essentially the connections between different router ports. These adjacencies facilitate the transport of packets among different network segments, such as between various metropolitan areas. The interfaces on these routers over which adjacencies are configured include, e.g., Ethernet or Optical Transport Network interfaces. [9, p. 32]
In addition to adjacencies, routers in the IP backbone also participate in Autonomous Systems (ASes). An AS is a collection of networks that operate under a single administrative domain and share a common routing policy. Each AS is assigned a unique Autonomous System Number (ASN) that identifies it on the Internet. Routers within an AS communicate with each other using an Interior Gateway Protocol (IGP) such as OSPF or IS-IS. When a packet needs to be routed outside the AS, it is forwarded to a router, which communicates with other ASes using an Exterior Gateway Protocol (EGP) such as Border Gateway Protocol (BGP), thereby knowing where to forward the packet. By using ASes, the Internet can be organized into a hierarchical structure that facilitates efficient routing and management of network traffic. [9, pp. 42–43, 45]

In summary, the IP backbone is a sophisticated and vital component of ISP networks, distinguished by its hierarchical structure, efficient data transport mechanisms, and focus on service availability. It operates at the core of the ISP network, connecting different metropolitan areas and managing high data traffic volumes. The backbone's design, incorporating various router types and connection methods, reflects the complex requirements of modern digital communication and the Internet's global nature.

3.3 CURRENTLY USED CONFIGURATION-METHODS

Currently used configuration methods at *Deutsche Telekom* include the manual configuration over the CLI, the NDO, and other systems that offer the

configuration of a part of the different networks. The following paragraph will name problems with these methods.

The manual configuration over the CLI introduces human error to the configuration process. Misconfiguring a router can lead to all sorts of connection issues, like BGP misconfiguration leading to inter-AS-communication failures or interface misconfiguration leading to route flapping or complete failure of the link. To assess this problem, one could argue to train network operators even more to minimize the risk of human error, but the better approach is to replace the human interaction with an automatic system [20, p. 246].

The NDO and other systems that are used to configure the IP backbone might not be as prone to error as the manual configuration over CLI, but the fact that they are multiple systems contradicts the idea of having a SSoT. If the configuration of the whole network is not just dependent on manual changes but also on multiple other systems, the only way of acquiring the current state is to continuously fetch the data from the network devices since they must have the whole configuration to run[1]. This situation makes different things harder, e.g., centralized control over the network, update automation, and recovery of multiple devices, as configurations need to be applied on numerous devices.
Having a SSoT can help because it is a form of centralization that enables people or scripts to overlook multiple configurations at once, make needed changes in one place, and finally deploy them automated at once. [17, pp. 8–9]

3.4 SDN

To tackle the problematic intricacies of the previously mentioned configuration methods, a more automated and abstracted approach would be Software Defined Networking (SDN). One definition of a software-defined network is:

> *"A network in which the control plane is physically separate from the forwarding plane, and a single control plane controls several forwarding devices."* [17, p. 12]

A control plane is the part of a router that determines the way a network behaves by, e.g., using IGPs like OSPF or IS-IS to gather information about the network topology. It holds a Routing Information Base, which contains all the needed data for computing the best paths through a network [17, p. 4].
A forwarding plane or data plane receives the computed information from the control plane and uses it to forward data packets. The so-called Forwarding Information Base contains the necessary information to forward a packet to the correct switch port so that it eventually reaches its destination [17, p. 4].

1 Assuming the control plane and the data plane both are on the network device

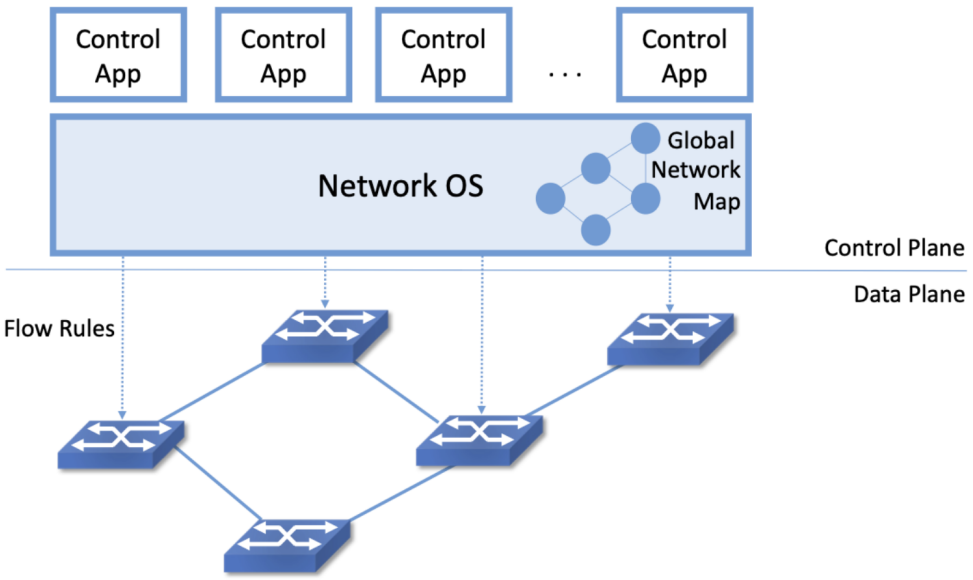

Figure 3.1: SDN architecture [17, p. 8]

One reason to separate these planes is to centralize the control plane of multiple devices logically. This enables the control plane to compute a globally optimized solution for the entire network regarding forwarding rules. It also removes the need for IGPs because the single devices no longer need the knowledge over the network. They get their forwarding rules not by computing them by themselves but from the centralized control plane [17, pp. 8–9].

The previously mentioned separation of control and data plane is displayed in fig. 3.1. One important fact to point out is that the control plane is portrayed as a distributed system. This design decision will increase availability and resistance against a total system outage [17, p. 10].

The element between the network hardware and the controller instances is the so-called Network OS. It contains a global network map, an abstracted model of the actual network topology. It is built and modified by information from the network devices and enables the controllers to manage the whole topology more easily. The controller can, e.g., edit the global network map to change the abstracted network model, which is then translated into changes to the actual network. When these changes are deployed, the network operators don't have to access the devices themselves and make manual changes but edit a more abstract version of the devices, and the final changes are made by an application that translates these changes for the network devices [17, p. 8].

4 CONSISTENCY IN NETWORKS

Changes appear relatively frequent in large carrier networks like IP backbones [9, pp. 255, 256]. These may occur due to planned reasons like maintenance or unplanned reasons like network failures. Despite the best efforts of network operators, network disruptions can still occur due to various reasons, such as hardware or software failures, human error, or external factors like natural disasters or cyber-attacks. Still, a device may sometimes be configured differently than expected due to manual configuration changes or software bugs. The network has to handle these disruptions since it is expected to be highly reliable and available.

In a network environment where updates are managed across multiple devices, challenges arise due to the distributed nature of the process. Imagine updating software or configurations on a series of network devices, such as routers, switches, or firewalls. Each device operates independently, yet they must coordinate for a consistent network operation. If an update fails on one device while others succeed, this can lead to network inconsistencies or even failures. Therefore, ensuring that all devices are either successfully updated or none are is crucial. As the next sections will explain, there are different ways to handle the challenges of distributed changes.

4.1 TRANSACTION

A transaction is an essential concept in database systems, forming the backbone of reliable and consistent data management. As perceived by an application programmer, the logical database typically functions within a relational database model, where application programs interact with data tuples through an Structured Query Language (SQL) interface. Transactions in this context are sequences of read and update actions performed on the logical database based on application requests. These sequences of actions are atomic, meaning that their effects are either entirely recorded in the database, resulting in the transaction being committed, or not recorded at all, leading to the transaction being aborted and rolled back. [21, p. 5]

A transaction in a relational database is conceptualized as an action sequence that may consist of a forward-rolling phase, possibly followed by a backward-rolling phase of undo actions. The forward-rolling phase can include partial rollbacks. This transaction model is vital in various discussions and analyses of transaction processing. A typical database server in a relational database management system functions as a transaction server, managing these transactions. [21, pp. 7, 8]

Central to understanding the behavior and role of transactions in databases are the ACID properties: Atomicity, Consistency, Isolation, and Durability.

1. **Atomicity**: This property ensures that all updates performed by a committed transaction are reflected in the database, while all updates by an aborted transaction are completely rolled back. Essentially, a transaction is an indivisible unit of work, and its operations must all occur or not occur at all.

2. **Consistency**: This property mandates that every committed transaction must preserve the consistency of the database. If a database starts in a consistent state, the transaction should not violate any integrity constraints, maintaining the overall consistency.

3. **Durability**: Durability guarantees that once a transaction is committed, its effects are permanently recorded in the database. This persistence remains even in cases of system failures, ensuring that the changes made by the transaction are not lost.

4. **Isolation**: The isolation property gives each transaction the impression that it is operating independently, without interference from other concurrent transactions. It controls how and when the changes made by one transaction become visible to others, and it ensures that concurrently running transactions do not lead to inconsistent database states.

The management of these ACID properties is a collaborative effort between the database management system and the application programmer. While the database system typically manages atomicity and durability, consistency and isolation require careful handling by both the system and the application programmer. For instance, the programmer must ensure that each transaction, when executed in isolation, maintains the database's consistency. Meanwhile, the database system is responsible for preserving this consistency across multiple concurrent transactions, which is particularly challenging and crucial for maintaining the integrity of the database. The level of isolation can be adjusted, with higher levels providing more robust guarantees of consistency at the potential cost of reduced concurrency and performance. [21, pp. 10–12]

In practical terms, transactions are fundamental in various real-world scenarios where data integrity and consistency are critical. From financial systems that handle sensitive monetary transactions to e-commerce platforms managing customer orders and inventory systems tracking stock levels, the principles of transactions are applied to ensure data accuracy, prevent data loss, and maintain a consistent state of the database despite concurrent access by multiple users or applications.

4.2 DISTRIBUTED TRANSACTIONS

Distributed transactions are a complex and intriguing topic expanding upon the traditional concept of transactions in relational databases. In this context, transactions extend beyond a single database, enclosing multiple databases within a distributed system. This expansion is not just a scaling of operations but introduces new challenges and considerations. [21, p. 299]

In a distributed transaction, the sequence of actions initiated by an application spans several databases. This means that a single transaction can include operations on data living in different databases, thereby complicating the management of ACID properties. The distributed nature of these transactions determines a robust coordination mechanism across various servers to ensure atomicity, consistency, isolation, and durability are maintained across the entire transaction, not just within a single database. [21, p. 300]

One of the primary challenges in distributed transactions is achieving *atomic commitment*. The goal here is to ensure that all parts of the transaction either commit or none do, preserving the integrity of data across the system. This challenge is commonly addressed by implementing protocols like the two-phase commit (section 4.2.1), which provide a framework for ensuring that all components of a distributed transaction reach a consensus on whether to commit or abort. [21, pp. 306–310]

Additionally, distributed databases grapple with issues such as data replication, which is employed to enhance availability and performance. However, replication introduces complexities in maintaining data consistency, particularly during updates. Various strategies are employed to manage these complexities, including different replication techniques and consistency models. [21, p. 314]

Failures and recovery also play a significant role in managing distributed transactions. Strategies for dealing with server failures, data recovery, and maintaining high availability and reliability are critical. Techniques like the ARIES[1] algorithm, adapted for distributed environments, ensure the system can recover from failures without losing transaction integrity.[21, pp. 75, 76, 311, 312]
Furthermore, ensuring isolation and managing concurrency in a distributed database environment is significantly more challenging due to the involvement of multiple servers. The aim is to prevent conflicts and ensure that transactions do not interfere with each other, a task that becomes increasingly complex in a distributed setting. [21, pp. 312–313]

4.2.1 *Two-Phase-Commit*

The Two-Phase Commit (2PC) protocol is a fundamental algorithm used in distributed systems, particularly for managing distributed transactions to

1 Algorithms for Recovery and Isolation Exploiting Semantics

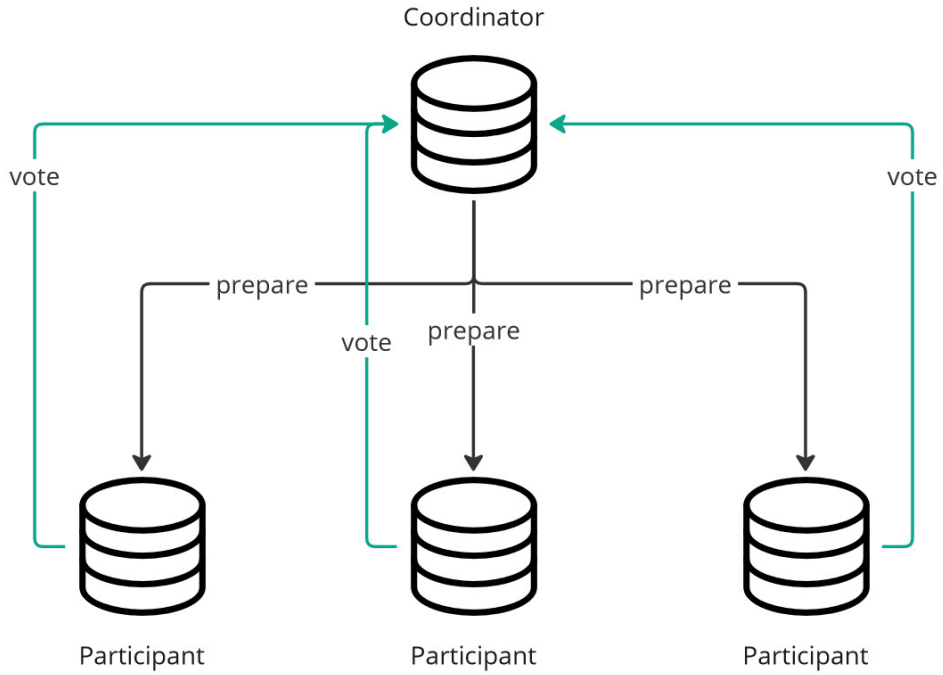

Figure 4.1: Vote collection phase

maintain data consistency and integrity across multiple databases. It is crucial in scenarios where a transaction involves multiple nodes or databases, and it's needed to maintain the atomicity aspect of the ACID properties. The essence of the 2PC protocol lies in its ability to provide a coordinated and fault-tolerant method to either commit or abort transactions in a distributed system. It involves two distinct phases: the prepare and commit phases, each serving a specific purpose in the transaction process. [21, p. 308]

In the prepare phase or vote collection phase (fig. 4.1), the transaction coordinator, typically a designated node in the system, initiates the protocol by sending a prepare message to all participant nodes involved in the transaction. This message is an inquiry to the participants, asking whether they are ready to commit the unit of work. Upon receiving this message, each participant node checks its local conditions to determine if it can commit the transaction. This might involve ensuring that it does not violate any constraints and that all the data required for the transaction is available and locked for exclusive access. If the conditions are favorable, the participant logs the transaction details to its storage and sends a "ready" message back to the coordinator. However, suppose the participant cannot commit the transaction (due to a constraint violation, for instance). In that case, it sends an "abort" message and undoes any changes. [21, p. 308]

The commit phase begins once the coordinator receives replies from all participant nodes. If all participants have sent a "ready" message, the coordinator commits the transaction. It then sends a "commit" message to all participants, instructing them to make the transaction changes permanent. Upon receiving the commit message, each participant commits the transaction locally, releases any locked resources, and sends an acknowledgment to the coordinator. However, if any participant has sent an "abort" message during the preparation phase, the coordinator sends an "abort" message to all participants. The participants then roll back any changes made during the transaction and release their locked resources. [21, p. 308]

On the one hand, the 2PC protocol effectively addresses several challenges of distributed transactions. Firstly, it ensures atomicity, as it guarantees that either all nodes commit the transaction or none do. This is crucial in maintaining consistency across the distributed system. Secondly, it provides a structured way to handle failures. For instance, if a participant node crashes after sending a "ready" message but before the commit, the coordinator can still decide to commit the transaction, and the participant can recover and complete its part of the transaction when it comes back online. Similarly, if a coordinator crashes after sending the commit message, participants can decide to commit the transaction independently after a timeout. [21, pp. 307, 311–312]

On the other hand, the 2PC protocol has its drawbacks. It is a blocking protocol, meaning that if the coordinator crashes after sending the prepare message but before sending the commit/abort message, the participant nodes can be left in an uncertain state, waiting indefinitely for the coordinator's decision. This can lead to resource-locking issues and affect the availability of the system. Furthermore, the protocol requires all participants to be able to communicate with each other and the coordinator, making it vulnerable to network partitioning issues. [21, p. 313]

4.2.2 *Sagas*

As Hector Garcia-Molina and Kenneth Salem conceptualized in their seminal 1987 paper, sagas represent a fundamental shift in managing long-lived transactions Long Lived Transactions (LLTs) in distributed systems. These LLTs, due to their extended duration compared to standard transactions, often cause significant performance issues in database systems. Sagas offer a novel approach to mitigate these challenges. [11]

The core idea behind sagas is to decompose an LLT into a sequence of smaller, more manageable transactions. This decomposition allows other transactions to interleave with the smaller transactions of a saga, thus reducing the resource-locking time and improving overall system performance. Each transaction within a saga maintains the database's consistency, but the saga as a whole is treated as a non-atomic unit. This means that while each trans-

action is atomic, the saga can tolerate partial executions, which are not ideal but can be compensated for. [11, p. 250]

To handle partial executions, compensating transactions are used. For each transaction in a saga, a corresponding compensating transaction is defined. This compensating transaction semantically undoes the effects of the original transaction but does not necessarily revert the database to the exact state before the transaction's execution. For instance, if a transaction in a saga books a flight seat, its compensating transaction would cancel that reservation. Figure 4.2 displays the local transactions A_1 to A_3 with their corresponding counter transactions B_1 to B_3. If A_3 fails, first B_2 and then B_1 are executed to reverse the finalized transactions A_1 and A_2. This approach allows sagas to maintain a level of consistency even in partial executions, which is crucial in distributed systems where failures and partial completions are common. [11, p. 250]

Several key differences emerge when comparing sagas to the 2PC protocol. While 2PC ensures atomicity and consistency, it can lead to significant performance bottlenecks due to its lock-based approach and the need for all participants to agree on the transaction's outcome (section 4.2.1).
Sagas, on the other hand, offer a more flexible model, especially suited for LLTs in distributed environments. They can reduce locking delays and improve concurrency by breaking down a large transaction into smaller ones and using compensating transactions. This design choice makes sagas particularly effective in scenarios where LLTs are common and system performance is a priority. However, the trade-off is in the complexity of managing compensating transactions and ensuring that they correctly undo the effects of their corresponding transactions. [11, p. 250]

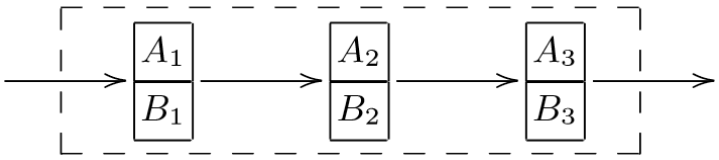

Figure 4.2: A sequential saga [3, p. 210]

5 IMPLEMENTATION

The implementation of the prototype consists of two main parts: the virtual router-lab 5.1 and the script that executes the changes 5.2.

5.1 VIRTUAL ROUTER-LAB

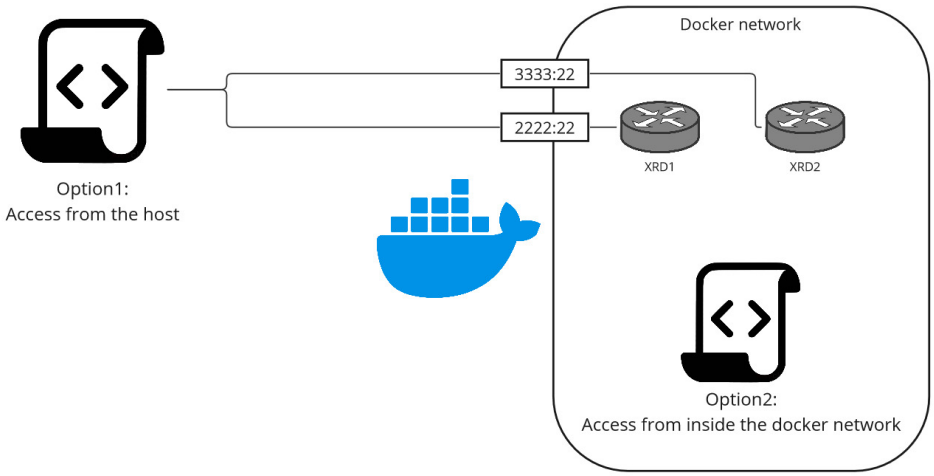

Figure 5.1: Router lab architecture

As fig. 5.1 shows, the router lab is built with Docker, a virtualization software that allows so-called "containers" to run isolated on a host machine [7]. These containers are instances of Docker images and one way to build images is to write a Dockerfile. This Dockerfile contains instructions on how to make an image, like which base image should be used, e.g., `alpine:3.19.0`, which is a lightweight Linux distribution. Other possible entries might copy files or folders from the host machine into the image or install dependencies for an application. An example can be found in the appendix listing A.3. There are package registries like https://hub.docker.com/ that host many images for different applications that can be pulled to the local Docker environment. Using the Docker image, an instance of itself can be started as a Docker container.

The reason to use Docker instead of, e.g., virtual machines in this implementation is that it can start and remove multiple router containers fairly quickly and can run on almost all devices. Quick startup times of containers are possible because Docker uses the already loaded and started host OS with its

libraries. This also enables the host machine to run many containers at once, while VMs use much more resources.

A VM has its own Operating System (OS) that runs on allocated VM resources. This setup does not scale well, as five VMs mean five loaded OSes that run separately and use many resources. In comparison, containers share the host's OS/kernel and, therefore, are more scalable horizontally. A visual representation of this difference can be seen in figs. 5.2 to 5.3 [18, p. 75f].

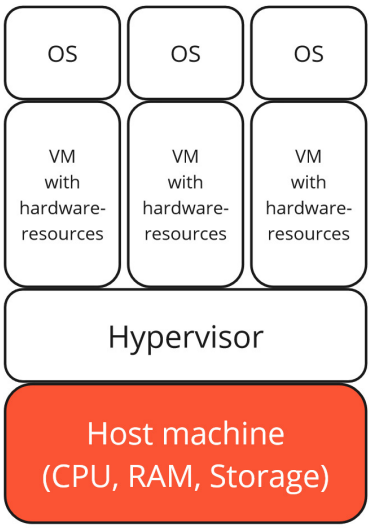

Figure 5.2: Virtual machine model

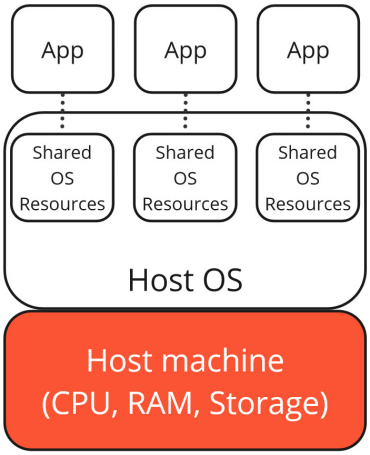

Figure 5.3: Container model

To host the router-lab, a host machine with enough resources is needed. In this case, a virtual machine is used on top of a VMware hypervisor with 16 CPU cores and 32 GB of RAM. The operating system is `Ubuntu 23.04`, and the needed dependencies are Docker and Docker Compose to run the virtualized routers.

5.1.1 *Virtual router*

In the beginning, the router-lab will only contain Cisco's Xrd-router instances[1] since they compare to routers used in the IP backbone of the *Deutsche Telekom*. The download from Cisco contains a Docker image, which can be imported into the local Docker environment. One thing to mention is that the used software[1] includes the image `xrd-control-plane:7.9.1`, which has essential packet forwarding capability and focuses on providing a system that targets compute-heavy tasks like "Virtual Route Reflector" and "Path Compute Element" [6]. To use these virtual routers in a production environment,

1 The used software can be found at:
https://software.cisco.com/download/home/286331236/type/280805694/release/7.9.1?i=!pp

the image `xrd-vrouter` would be used, which allocates more memory and needs at least two dedicated CPU-cores to provide a highly performant data-plane [6].

After importing the `xrd-control-plane` image, the router can be spun up using Docker or using scripts from an open-source-project called xrd-tools 5.1.2.

5.1.2 *xrd-tools*

The repository https://github.com/ios-xr/xrd-tools comes with helpful scripts that especially make the usage of multiple router instances easier. The first script used when setting the host machine up was `host-check`, which scans the machine's configuration and informs about needed tweaks to enable the routers to run flawlessly. It checks for requirements like CPU cores, free RAM, installed modules, and configured kernel parameters. Proposed changes are listed in the script's output.

In this specific case, the maximum number of *Inotify* instances, which are used to notify a program about changes in the filesystem, needed to be increased:

```
FAIL -- Inotify max user instances
        The kernel parameter fs.inotify.max_user_instances is set to 128 but
        should be at least 4000 (sufficient for a single instance) - the
        recommended value is 64000.
        This can be addressed by adding 'fs.inotify.max_user_instances=64000'
        to /etc/sysctl.conf or in a dedicated conf file under /etc/sysctl.d/.
        For a temporary fix, run:
          sysctl -w fs.inotify.max_user_instances=64000
```

Figure 5.4: host-check output

As the script output suggests, a temporary fix would be to set the currently allowed maximum inotify instances per user with `sudo sysctl -w fs.inotify.max_user_instances=64000` or add a configuration file under `/etc/sysctl.d/` with the content `fs.inotify.max_user_instances=64000`. Setting a value of 64000 allows 16 router instances to run simultaneously, which will be enough for the prototype.

Once every needed change was applied, the first instance was started using the `launch-xrd` script, which constructs a Docker command that runs a router container (fig. A.1).

5.1.3 *Custom xrd image*

The `xrd-control-plane` is a basic router image without additional configuration. A custom image was built to activate the management interface by default, fetch its IPv4 address via Dynamic Host Configuration Protocol (DHCP), and activate the local SSH server (listings A.1 to A.2). Using this image and forwarding port 22 from inside the router container to the host

system makes the router accessible to the user via SSH. This image is mainly used while setting up the lab because the routers that will be tested will have their own configuration, so they will start as plain `xrd-control-plane` containers and get their current running configuration applied over SSH or initially start with the wanted configuration.

5.1.4 *Virtualized topologies*

With the previously mentioned steps a single instance can be started and used for testing the change executor script (section 5.2). The next goal is to enable the testing environment to run multiple routers at once to create a virtualized network topology. When the routers start-up, their interfaces are put into different networks to simulate a link. In this way the routers communicate with each other and, if routing protocols are set up, with other not directly connected devices.

To achieve this functionality, *Docker Compose* is used since it lets the user define a topology of Docker containers that are started together and put into user-defined networks. To make Docker Compose work with the virtual routers, another script of the xrd-tools repository (section 5.1.2) is used: `xr-compose`. This script translates an extended version of the Docker Compose syntax into the plain and expected YML syntax. The extended version includes, e.g., a simpler way to define links between routers. After translating an "xr-specific" file to a plain compose file, Docker Compose can start the predefined topology.

To test the Docker Compose setup, a network topology[2] has been defined as displayed in fig. 5.5. In total five routers and two Linux containers are started to simulate an MPLS core network and an access network. The goal is that the Linux containers can ping each other, confirming that the routers established a working connection. The access network and the core network are in different OSPF areas, and the core is set up to use MPLS. XRD2 and XRD4 have a LSP over XRD3. Since the access routers XRD1 and XRD5 are set up to advertise their networks to the core-edge routers XRD2 and XRD4 and vice versa, the Linux containers can reach each other by a default route to their router.

2 This topology is inspired by https://www.rogerperkin.co.uk/ccie/mpls/cisco-mpls-tutorial/

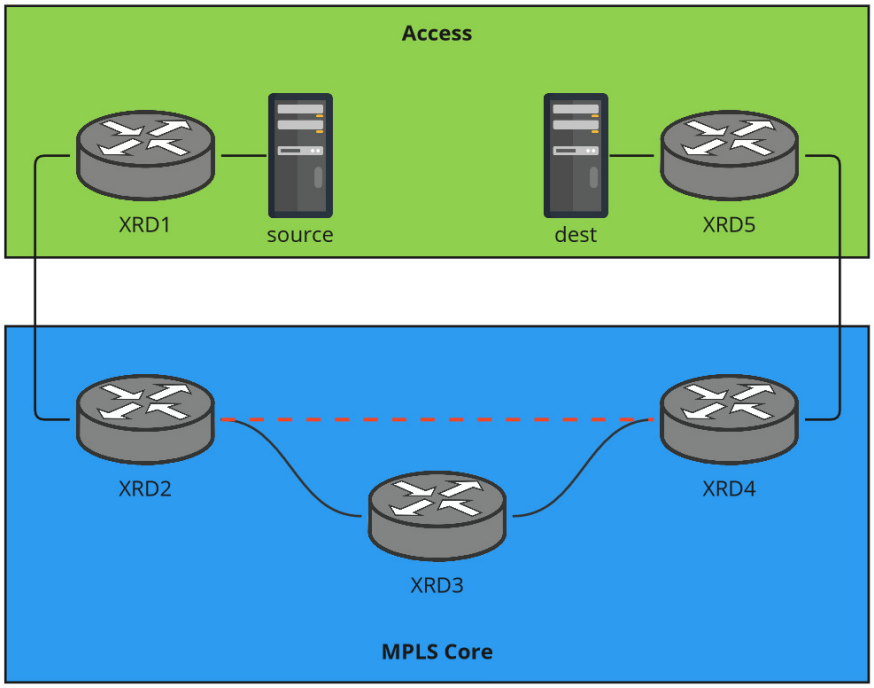

Figure 5.5: MPLS topology

This expected behavior has been tested on the router lab host machine. The codebase can be found in https://github.com/dominikmandok/hda-thesis/tree/main/xr-compose-mpls. After starting the Docker Compose topology and waiting less than a minute, the routing table 5.1 has formed at the XRD1 router. Figure 5.6 shows a traceroute from the *source* container to the *dest* container, using six hops, confirming the reachability between the Linux containers.

```
ubuntu@ubuntu:~$ docker exec -it source /bin/sh
/app # traceroute 10.10.3.3
traceroute to 10.10.3.3 (10.10.3.3), 30 hops max, 46 byte packets
 1  xr-1.xr-compose-mpls_source-xrd-1 (10.10.0.3)  0.692 ms  0.403 ms  0.371 ms
 2  10.10.1.2 (10.10.1.2)  1.003 ms  0.837 ms  0.720 ms
 3  10.11.0.2 (10.11.0.2)  2.501 ms  2.340 ms  2.134 ms
 4  10.11.0.6 (10.11.0.6)  2.124 ms  2.067 ms  2.127 ms
 5  10.10.2.2 (10.10.2.2)  2.072 ms  2.148 ms  2.054 ms
 6  10.10.3.3 (10.10.3.3)  2.145 ms  2.106 ms  2.151 ms
/app #
```

Figure 5.6: MPLS topology traceroute

Table 5.1 shows that XRD1 knows all the other routers' Loopback addresses indicated by the numbers, e.g., 2.2.2.2 for XRD**2**. All the routes with the type "O IA" have been learned via OSPF from another OSPF-area, thus "**i**nter-**a**rea". The types "L" and "C" stand for "local" and "connected" and are known since the router started. This result shows that the wanted network topology

Type	Network	Next Hop	Interface
L	1.1.1.1/32	Direct	Loopback0
O IA	2.2.2.2/32	10.10.1.2	GigabitEthernet0/0/0/1
O IA	3.3.3.3/32	10.10.1.2	GigabitEthernet0/0/0/1
O IA	4.4.4.4/32	10.10.1.2	GigabitEthernet0/0/0/1
O IA	5.5.5.5/32	10.10.1.2	GigabitEthernet0/0/0/1
C	10.10.0.0/24	Direct	GigabitEthernet0/0/0/0
L	10.10.0.3/32	Direct	GigabitEthernet0/0/0/0
C	10.10.1.0/30	Direct	GigabitEthernet0/0/0/1
L	10.10.1.1/32	Direct	GigabitEthernet0/0/0/1
O IA	10.10.2.0/30	10.10.1.2	GigabitEthernet0/0/0/1
O IA	10.10.3.0/24	10.10.1.2	GigabitEthernet0/0/0/1
O IA	10.11.0.0/30	10.10.1.2	GigabitEthernet0/0/0/1
O IA	10.11.0.4/30	10.10.1.2	GigabitEthernet0/0/0/1
C	172.30.0.0/24	Direct	MgmtEth0/RP0/CPU0/0
L	172.30.0.1/32	Direct	MgmtEth0/RP0/CPU0/0

Table 5.1: Routing Table at XRD1

has successfully been created and all the expected connections have been established.

With the router lab working and the capability to start single routers and whole network topologies, the next task is to build the script that tests configuration changes on these virtual routers.

5.2 CHANGE EXECUTOR SCRIPT

With the virtual lab in place, the development of the change executor starts. This system is responsible for updating router configurations in the lab while assuring network consistency. The programming language used to code the prototypes' solution is Python, as it comes with many additional packages that can easily be used in one's projects [19], e.g., *netmiko* (section 5.2.2) or *jsonschema*.

The following figure shows the first architectural idea of the whole BBaC project as already described in section 1.1:

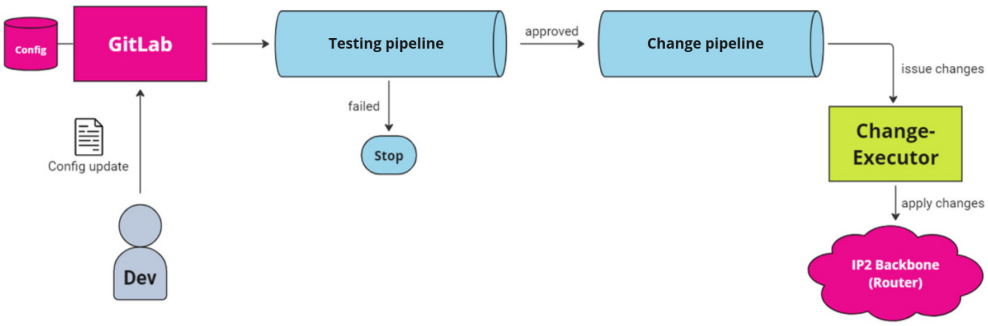

Figure 5.7: BBaC architecture

After the GitLab instance, the first step is a pipeline that should check if specific changes might cause problems in the IP backbone. The change executor script is used not only when applying changes in the actual IP backbone, but also in the testing pipeline to simulate the change on a virtual simulation of selected routers (section 5.1).

The sequence of steps is meant to be as follows:

1. Read the planned changes from a job-file
2. Backup the running configurations of the devices
3. Execute the changes
4. If errors occur, roll back any changes

A job file is a JSON file with information about which device changes must be made. This includes, e.g., credentials for the router, the path to the new running-configuration, or the IP address and port of the router (listing A.5). It has a forced structure that is defined by three files[3]. The three provided files define the so-called *schema* of a job file. It is possible to automatically validate the syntax of a given job file with the provided schema file in Python with the *jsonschema*[4] library. The change executor uses this step to ensure all needed information is provided to the script so that it can be executed.

3 They can be found at:
https://github.com/dominikmandok/hda-thesis/tree/main/schemas
4 https://pypi.org/project/jsonschema/

After validating the job file, the script backs up the currently running configuration of the network devices. This step uses the *netmiko* library (section 5.2.2) to connect and send commands to the devices. After all devices have been backed up, the changes get applied. This is also done using the established connections via *netmiko*. If a router does not accept the new configuration, an exception is thrown and the rollback mechanism starts.

The rollback mechanism checks if the previously saved backup configurations are still present and reuploads these configs. An important step is actually to overwrite the current configuration instead of reapplying it. The details are further described in section 5.2.2.

5.2.1 *The Docker image*

A new Docker image is built to ensure every script dependency is installed and correctly configured (listing A.3). It extends the lightweight Docker image `alpine:3.19.0` with source folders containing the change executor script and schema-files, which define JSON-schemas used later in the script. The last steps are to create a virtual environment and install all the dependencies defined in a "requirements.txt" file. With these dependencies and the source files, the image can be used in the testing environment.

5.2.2 *Netmiko*

As the change executor has to connect to routers to apply changes, a Python library called *netmiko*[5] is used. This library provides predefined classes for router models that handle certain functions differently. There might be certain methods implemented for one set of router models and other methods that are not implemented, and the console output is parsed using specific regular expressions. It also supports multiple connection methods like Telnet or SSH.

Since the implementation is a prototype that only uses virtual Cisco XR routers and SSH is the preferred communication method, the script uses the `CiscoXrSSH`-class. `CiscoXrSSH` inherits from the `CiscoXrBase`-class which overrides multiple methods. One reason to fork the *netmiko* library and change the source code is a missing feature in the `commit()` method. This missing feature and the necessity to have it in the library for this special case is explained in the following.

The default behavior of the "commit" command in the used router software is to "commit the target configuration to the active (running) configuration" [5]. This means, e.g., if the current running configuration has no hostname

5 The following link references a forked version of *netmiko* that has been extended with project-specific capabilities:
https://github.com/dominikmandok/netmiko/tree/adding_commit_replace_command

configured and the hostname gets set in a configuration session, the "commit" command will apply the hostname to the running configuration:

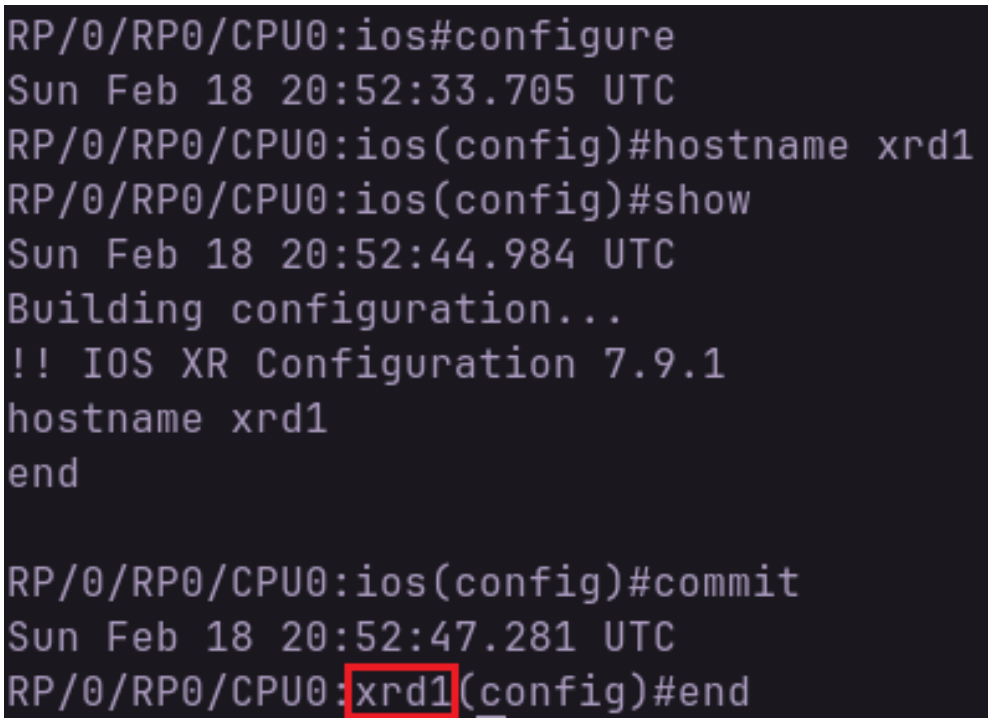

Figure 5.8: Commit a new hostname

After the commit, the hostname changed in the console window from "ios", which is the default if no hostname is set, to "xrd1".

It's important to mention that this "commit" action applies every entry in the "target configuration" to the "running configuration". Meaning if a script saves the old running configuration without any hostname attribute as a backup, applies a new configuration with the hostname attribute set, and then tries to apply the backup config to roll back any changes, the hostname will not be reverted to "ios", because the absence of a keyword in the "target configuration" will not reset it in the "running configuration". This is a problem since the router is in a different state after the rollback than before the changes, although the change executor has to provide network consistency. To ensure that the backup config overwrites the current running-config entirely, the command "commit replace" has to be used, which "Replaces the entire running configuration with the contents of the target configuration" [5].

The *netmiko* library does implement the described default behavior of the "commit" command, but it does not implement the "commit replace" command. For this reason, the needed code was added to the library's fork, which is used in the change executor. The changes can be found in [15] as well as in the appendix (listing A.4). In this state, the *netmiko* library is ready to be used in the change executor.

5.2.3 *Testing*

Testing the source code and the running application is crucial to prevent failures of the change executor in the production environment. To address this need, different tests have been written. The system running these tests is GitLab CI/CD, an integrated part of the GitLab platform used for Continuous Integration and Delivery. It automates the process of building, testing, and deploying applications. Developers can use it to continuously run tests on their code changes, ensuring the application's reliability and stability during development [12].

5.2.3.1 *Static testing*

The static tests include GitLab Static Application Security Testing (SAST)[6]. SAST scans the repository for source code files and detects the different languages. It then automatically uses the corresponding scanners to search for known security concerns in the source code. The findings help secure the script from possible exploitation, making it more reliable.
Another measure in static testing is self-written tests, verifying the code's functionality. This includes checking object attributes for correct values after the object is used or comparing a method's return value to an expected result. Assertions will crash the program, if unexpected results happen, resulting in a failed CI/CD pipeline (fig. 5.10).

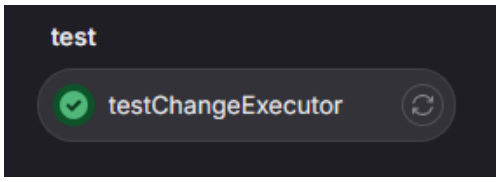

Figure 5.9: Tests succeded

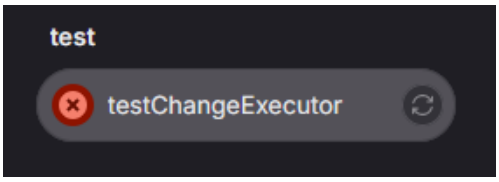

Figure 5.10: Tests failed

An excerpt of the static tests can be found in listing A.6.

5.2.3.2 *Dynamic testing*

Two dynamic test cases have been created to continuously test the behavior of the change executor in development. Instead of testing just the code, they examine the change executor's behavior in a live environment. This means two routers will be started, and one Linux container running the change executor will be put inside the same Docker network as the routers (fig. 5.1 option 2).

Test case one sets up a situation where the routers have a small base configuration to enable the SSH server since the change executor communicates over SSH. The script will try to update both routers' configuration in one job,

6 More info at:
https://docs.gitlab.com/ee/user/application_security/sast/

which is expected to work flawlessly. The job file looks like listing A.5 and the referenced configuration file is for both changes listing 5.1. Both updates must be executed without errors, or the script does not work as intended.

```
hostname xrd
```

Listing 5.1: Correct configuration

```
wrong_input will fail
```

Listing 5.2: Wrong configuration

The second test case checks the rollback mechanism of the change executor. The initial setup is the same as in test case one: two routers are started with a basic configuration to enable the SSH server and one Linux container running the script. What differs is the used job file. Instead of using the same referenced new configuration for both routers, it uses the correct one (listing 5.1) for the first router and the wrong config (listing 5.2) for the second one. When executing the script, the stages as described in section 5.2 will run: First, the script backs up the running configurations of the routers. After successfully saving the two files, the change executor starts applying the new configurations (listings 5.1 to 5.2). The first update is applied on router one without errors and gets committed to its running configuration. When updating the second router, it will return an error message informing the script about invalid input. At this point, it throws an exception and starts the rollback phase. The script checks if the backed-up configuration files are still present and sequentially applies them back to the routers. This special case highlights the importance of the changes made to the *netmiko* library, which are described in section 5.2.2.

These test cases cover the change executor's basic functionality and, therefore, will prevent flawed scripts from being deployed.

6

CONCLUSION

As this thesis concludes, it is important to contextualize the BBaC project within the larger scope of research on network management. While BBaC is a part of the broader research initiative, the focus extends beyond this project to encompass the broader implications and applications of distributed transactions and sagas in complex network systems.

DISTRIBUTED TRANSACTIONS IN NETWORK MANAGEMENT

As introduced by Hector Garcia-Molina and Kenneth Salem, the concept of distributed transactions and sagas offers a fundamental shift in managing LLTs in distributed systems [11], such as those encountered in large-scale network management (section 4.2.2). These systems, characterized by their extended durations and complex interdependencies, often face significant performance issues [11, p. 249]. With their novel approach of decomposing a single long transaction into a series of smaller, more manageable transactions, sagas introduce a new way of handling these complexities. This methodology enhances system performance by reducing resource-locking time and maintains the network's consistency, a critical factor regarding network availability and functionality.

APPLICABILITY TO BROADER NETWORK MANAGEMENT

The BBaC project, while an essential element of this thesis, serves as a practical application of these concepts. The project's goal of transforming the configuration and management of routers within an IP backbone (section 1.1) aligns well with the principles of distributed transactions and sagas (section 4.2). As a network management system solely responsible for a whole backbone introduces the possibility of complex configuration tasks, a solution to these LLTs must be developed. By breaking down complex network configuration tasks into smaller, manageable transactions, BBaC can efficiently manage updates and changes across the network. The sagas' approach, especially the use of compensating transactions to handle partial executions and maintain consistency, is particularly relevant. This method allows for flexibility in a network's operation, ensuring that even in cases of partial failures, the system can revert to a consistent state, thereby minimizing disruptions and maintaining network integrity.

IMPLEMENTATION AND FUTURE DIRECTIONS

Reflecting on the implementation aspects covered in this thesis, including the development of a testing pipeline and a prototype system (chapter 5), it becomes evident how distributed transactions and the saga methodology can significantly contribute to the efficiency and reliability of network management systems that provide an SSoT regarding network device configurations and must persevere network consistency. These concepts are not just limited to the scope of the BBaC project but have broader applicability in various domains where distributed systems play a pivotal role. The insights gained from this research can influence future advancements in network management, pushing the boundaries of how network configurations are handled, tested, and maintained (chapter 7).

7 FUTURE WORK

A basic prototype of a testing pipeline was built with this thesis. It shows a possible way of updating router configurations and rolling changes back automatically if errors occur in the update process and demonstrates how to virtualize whole network topologies. The following sections will explain the next steps to further progress in implementing the BBaC project.

7.1 ENHANCING THE ROUTER LAB

As of now, the router lab (section 5.1) uses Cisco's XR router OS, which is sufficient for a prototype. To use the lab in a production environment, it has to support more router models from different vendors to simulate more topologies from the IP backbone. Any possible prerequisites for new models must be examined and implemented into the current lab's version. Adding multiple different supported router models is essential to providing a router lab that can extensively test real-world scenarios.

Another enhancement would be to automate the infrastructure the router lab runs on (IaC). Currently, it is a virtual machine that runs all the time, but an idea is to move the lab onto a virtual machine that gets spun up at need. With this approach, the VM's resources can be adjusted to meet the lab's needs. It would also be possible to start multiple labs that don't interfere with each other performance-wise, since they would run on different host machines.

7.2 CONFIGURATION ABSTRACTION

As described in section 1.1, the BBaC project uses GitLab to store router configurations. It is the network's SSoT, meaning it is a central place that dictates the live network's state. Currently, the router configurations are stored in their vendor and model native syntax. An idea is to abstract the configuration so that the same syntax can be used for different router models. A change like this enforces the usage of a translator that can transform the abstract language, which is used for every network device, into the native configuration syntax of the corresponding device. This translator adds complexity but also simplifies the configuration process for network operators since they don't need to know multiple systems at once.

A different way of reaching more abstraction is by developing an *intent-driven* approach to network configuration [16]. The thought behind intent-driven configuration is about identifying the network users' behavior, extracting information from it, and translating it to network configuration

changes. An intent is a "high-level policy used to operate the network" [1, p. 5]. This approach might be interesting since it requires network operators to instruct a behavior that the network management system translates into network device configurations, which can then be deployed.

7.3 IMPROVING USER EXPERIENCE

An enhancement regarding the user experience of the BBaC system would be to develop a different way to edit device configurations. The current idea is to store them in the native format, but as mentioned in section 7.2 an abstraction of the configuration language would make it easier to use the system. This idea can be extended to create a web frontend or an application that acts as an interface between the user and the plain configuration files. Pros of such an interface are the early prevention of errors through frontend validation and a faster and simpler change process.
A company-internal AI chatbot could also extend this frontend approach that may suggest configuration changes by a given user input. This could include describing an intent as mentioned in section 7.2 and receiving the needed changes.

7.4 MONITORING OF THE NETWORK

Assuming that the BBaC project will result in a tool with lots of influence and control over the backbone, a valuable extension is continuous monitoring of the network devices. An extensive alarming system can be set up to detect and report malfunctions as fast as possible using collected Key Performance Indicators (KPIs) like CPU usage, traffic per interface, or hardware temperature.

When such monitoring is set up, it allows the BBaC system to be somewhat self-healing. For example, if a link fails, the system might be able to reroute the traffic over another LSP or use a backup path while the problem of the failing link is addressed by a network operator or, if possible, also by the system itself. This allows for a shorter recovery time between the first appearance and the elimination of a failure.

Part II

APPENDIX

A IMPLEMENTATION

```
ubuntu@ubuntu:~/xrd-tools/scripts$ ./launch-xrd --privileged \
> --mgmt-interfaces linux:eth0,snoop_v4,snoop_v4_default_route \
> --name router \
> registry.devops.telekom.de/ml-automation/bbac_poc/xrd-control-plane:7.9.1 \
> --dry-run
docker run -it --rm --privileged --env XR_MGMT_INTERFACES=linux:eth0,snoop_v4,snoop_v4_default_route
--name router registry.devops.telekom.de/ml-automation/bbac_poc/xrd-control-plane:7.9.1
```

Figure A.1: `launch-xrd` script dry-run

```
username cisco
 group root-lr
 secret 10 ...
!
ssh server v2
```

Listing A.1: Custom XRD: every-boot.cfg

```
FROM localhost/xrd-control-plane:7.9.1
COPY every-boot.cfg /etc/xrd/
ENV XR_EVERY_BOOT_CONFIG /etc/xrd/every-boot.cfg
ENV XR_MGMT_INTERFACES linux:eth0,chksum,snoop_v4
EXPOSE 22
```

Listing A.2: Custom XRD: Dockerfile

```
FROM alpine:3.19.0

RUN apk update
RUN apk add openssh \
    python3 \
    py3-pip \
    git

RUN mkdir /schemas
RUN mkdir /test
RUN mkdir /app
WORKDIR /app

COPY ./app /app
COPY ./schemas /schemas
COPY ./test /test

COPY ./ping_test.py /app/ping_test.py

RUN python -m venv .
RUN source bin/activate && \
    pip install -r requirements.txt
```

Listing A.3: Change executor Dockerfile

```
class CiscoXrBase(CiscoBaseConnection):
    def commit(
        self,
        replace: bool = False   # This has been added for the
                                # "commit replace"-command
    ) -> str:
        # Select the command string based on arguments provided
        if label:
            pass
        # ...
        elif replace:
            command_string = "commit replace"
        # ...

        new_data = self._send_command_str(
            command_string,
            expect_string=r"(#|onfirm|Do you wish to proceed)"
            # ...
        )

        # "Do you wish to proceed" has to be checked, because
        # after sending the "commit replace" command,
        # the router asks for confirmation
        if ("onfirm" in new_data or
            "Do you wish to proceed" in new_data):
            output += new_data
            new_data = self._send_command_str(
                "y",
                expect_string=r"#"
                # ...
            )
        # ...
```

Listing A.4: *Netmiko* commit() method changes

```
{
    "changes": [
        {
            "host": "xrd1",
            "port": 22,
            "config_path": "/test/jobs/test_job_1.cfg"
        },
        {
            "host": "xrd2",
            "port": 22,
            "config_path": "/test/jobs/test_job_2.cfg"
        }
    ],
  "credentials": {
        "username": "cisco",
        "password": "ciscocisco"
    }
}
```

Listing A.5: Job file example

```python
def test_job_working() -> None:
    """Tests if a valid job-file is working correctly."""

    FILE_PATH = f"{CONFIG.JOB_DIR}/test_job_working.json"
    job1 = Job(FILE_PATH)

    assert (
        job1._ready == job1.is_ready()
    ), "is_ready() function is not working correctly"
    assert (
        job1._job_file_path == FILE_PATH
    ), "Job file path is not correct"
    assert (
        job1.is_ready()
    ), "Job is not ready, meaning the job-file is not valid"

    if not CONFIG.DRY_RUN:
        job1.execute()

        del job1  # kill the connection

    job2 = Job()

    assert (
        not job2.is_ready()
    ), "Job is ready, which can't be"
    assert (
        job2._job_file_path is None
    ), "Job file path is not None"

    job2.set_job_file_path(FILE_PATH)

    assert (
        job2.is_ready()
    ), "Job is not ready, meaning the job-file is not valid"
    assert (
        job2._job_file_path == FILE_PATH
    ), "Job file path is not correct"

    if not CONFIG.DRY_RUN:
        job2.execute()

        del job2
```

Listing A.6: Test of the job-class

BIBLIOGRAPHY

[1] M. Behringer, M. Pritikin, S. Bjarnason, A. Clemm, B. Carpenter, S. Jiang, and L. Ciavaglia. *Autonomic Networking: Definitions and Design Goals*. RFC7575. RFC Editor, June 2015, RFC7575. DOI: 10.17487/RFC7575. URL: https://www.rfc-editor.org/info/rfc7575 (visited on 11/13/2023).

[2] Martin Birk, Gagan Choudhury, Bruce Cortez, Alvin Goddard, Narayan Padi, Aswatnarayan Raghuram, Kathy Tse, Simon Tse, Andrew Wallace, and Kang Xi. "Evolving to an SDN-enabled isp backbone: key technologies and applications." In: *IEEE Communications Magazine* 54.10 (Oct. 2016), pp. 129–135. ISSN: 0163-6804. DOI: 10.1109/MCOM.2016.7588281. URL: http://ieeexplore.ieee.org/document/7588281/ (visited on 11/03/2023).

[3] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. "Theoretical foundations for compensations in flow composition languages." In: *ACM SIGPLAN Notices* 40.1 (Jan. 12, 2005), pp. 209–220. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/1047659.1040323. URL: https://dl.acm.org/doi/10.1145/1047659.1040323 (visited on 03/03/2024).

[4] Donald Cal, Seungjoon Lee, Shubho Sen, and Jennifer Yates. "Gold standard auditing for router configurations." In: *2010 17th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN)*. Metropolitan Area Networks (LANMAN). Long Branch, NJ, USA: IEEE, May 2010, pp. 1–6. ISBN: 978-1-4244-6067-0. DOI: 10.1109/LANMAN.2010.5507163. URL: http://ieeexplore.ieee.org/document/5507163/ (visited on 11/03/2023).

[5] Cisco. *Configuration Management Commands on Cisco IOS XR Software - Cisco*. URL: https://www.cisco.com/c/en/us/td/docs/routers/xr12000/software/xr12k_r3-9/system_management/command/reference/yr39xr12k_chapter5.html#wp1873946320 (visited on 02/18/2024).

[6] Cisco. *ios-xrd-ds.pdf*. 2023. URL: https://www.cisco.com/c/en/us/products/collateral/routers/ios-xrd/ios-xrd-ds.pdf (visited on 02/09/2024).

[7] Docker. *Docker overview | Docker Docs*. URL: https://docs.docker.com/get-started/overview/ (visited on 03/04/2024).

[8] Peter Lars Doral. *An Introduction to Computer Networks - Second Edition*. 2020. URL: http://intronetworks.cs.luc.edu/current2/ComputerNetworks.pdf.

[9] Robert D. Doverspike, K. K. Ramakrishnan, and Chris Chase. "Structural Overview of ISP Networks." In: *Guide to Reliable Internet Services and Applications*. Ed. by Charles R. Kalmanek, Sudip Misra, and Yang Yang. Series Title: Computer Communications and Networks. London: Springer London, 2010, pp. 19–93. ISBN: 978-1-84882-827-8 978-1-84882-828-5. DOI: 10.1007/978-1-84882-828-5_2. URL: http://link.springer.com/10.1007/978-1-84882-828-5_2 (visited on 02/02/2024).

[10] William Enck, Thomas Moyer, Patrick McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert Greenberg, Yu-Wei Sung, Sanjay Rao, and William Aiello. "Configuration management at massive scale: system design and experience." In: *IEEE Journal on Selected Areas in Communications* 27.3 (Apr. 2009), pp. 323–335. ISSN: 0733-8716. DOI: 10.1109/JSAC.2009.090408. URL: http://ieeexplore.ieee.org/document/4808476/ (visited on 11/03/2023).

[11] Hector Garcia-Molina and Kenneth Salem. "Sagas." In: *Proceedings of the 1987 ACM SIGMOD international conference on Management of data - SIGMOD '87*. the 1987 ACM SIGMOD international conference. San Francisco, California, United States: ACM Press, 1987, pp. 249–259. ISBN: 978-0-89791-236-5. DOI: 10.1145/38713.38742. URL: http://portal.acm.org/citation.cfm?doid=38713.38742 (visited on 02/22/2024).

[12] GitLab. *What is CI/CD?* URL: https://about.gitlab.com/topics/ci-cd/ (visited on 03/05/2024).

[13] Theo Haerder and Andreas Reuter. "Principles of transaction-oriented database recovery." In: *ACM Computing Surveys* 15.4 (Dec. 2, 1983), pp. 287–317. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/289.291. URL: https://dl.acm.org/doi/10.1145/289.291 (visited on 11/27/2023).

[14] Med Amine Harrabi, Maroua Jeridi, Noma Amri, Med Rabii Jerbi, Acem Jhine, and Houssem Khamassi. "Implementing NFV routers and SDN controllers in MPLS architecture." In: *2015 World Congress on Information Technology and Computer Applications (WCITCA)*. 2015 World Congress on Information Technology and Computer Applications Congress (WCITCA). Hammamet, Tunisia: IEEE, June 2015, pp. 1–6. ISBN: 978-1-4673-6636-6. DOI: 10.1109/WCITCA.2015.7367030. URL: http://ieeexplore.ieee.org/document/7367030/ (visited on 11/03/2023).

[15] Dominik Mandok. *Add support for commit replace in CiscoXrBase class · dominikmandok/netmiko@06be866*. GitHub. URL: https://github.com/dominikmandok/netmiko/commit/06be866f17e405209bde7cd972f4f6ebb34ad52e (visited on 02/18/2024).

[16] Kashif Mehmood, Katina Kralevska, and David Palma. "Intent-driven autonomous network and service management in future cellular networks: A structured literature review." In: *Computer Networks* 220 (Jan. 2023), p. 109477. ISSN: 13891286. DOI: 10.1016/j.comnet.2022.109477. URL: https://linkinghub.elsevier.com/retrieve/pii/S1389128622005114 (visited on 11/13/2023).

[17] Larry L. Peterson, Carmelo Cascone, Brian O´Connor, Thomas Vachuska, and Bruce Davie. *Software-Defined Networks: A Systems Approach*. Wroclaw: Systems Approach LLC, 2021. 189 pp. ISBN: 978-1-73647-210-1. URL: https://sdn.systemsapproach.org/.

[18] Nigel Poulton. *Docker deep dive: zero to Docker in a single book*. 2020 edition. OCLC: 1176226082. Germany: Nigel Poulton, 2020. ISBN: 978-1-5218-2280-7.

[19] *PyPI · The Python Package Index*. PyPI. URL: https://pypi.org/ (visited on 02/14/2024).

[20] J. T. Reason. *Human error*. Cambridge [England] ; New York: Cambridge University Press, 1990. 302 pp. ISBN: 978-0-521-30669-0 978-0-521-31419-0.

[21] Seppo Sippu and Eljas Soisalon-Soininen. *Transaction Processing: Management of the Logical Database and its Underlying Physical Structure*. Data-Centric Systems and Applications. Cham: Springer International Publishing, 2014. ISBN: 978-3-319-12291-5 978-3-319-12292-2. DOI: 10.1007/978-3-319-12292-2. URL: https://link.springer.com/10.1007/978-3-319-12292-2 (visited on 03/02/2024).